בס"ד

Introduction to Software Engineering

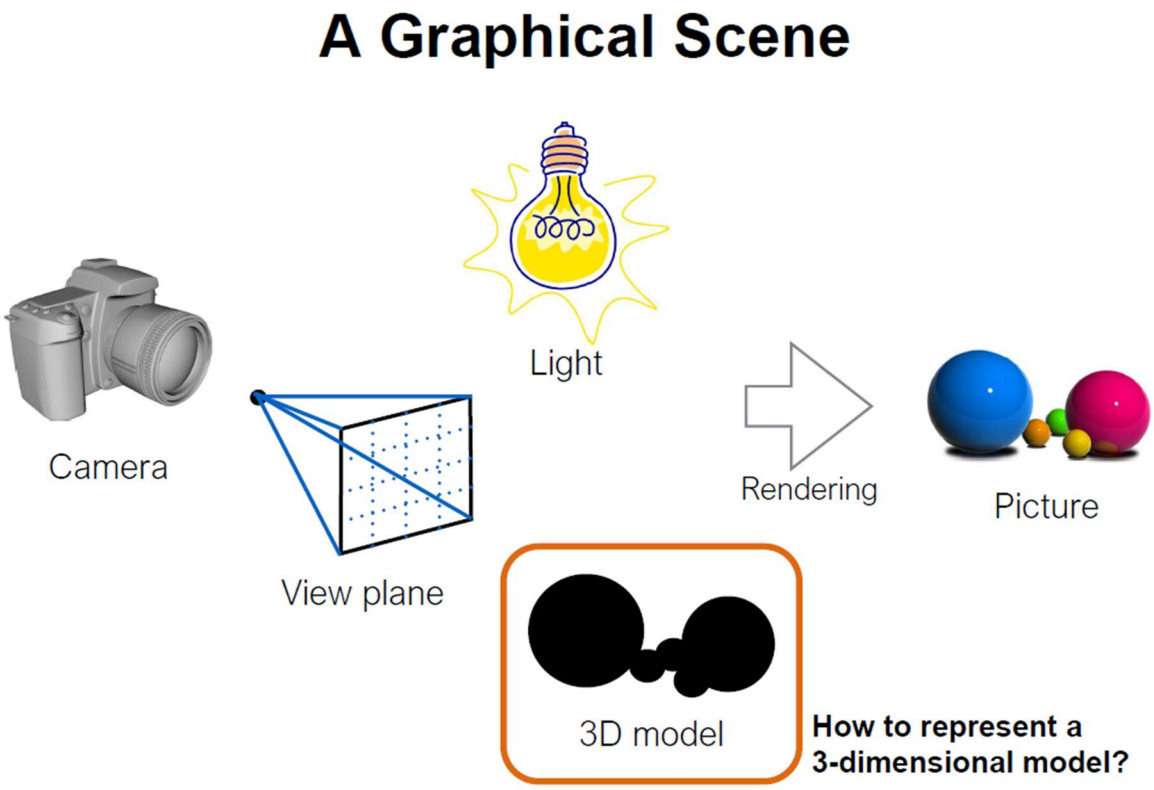
Mini Project

Chezki Botwinick
Yaakov Freidman

בס"ד

Introduction

The purpose of this project is to design and model a virtual 3-dimensional graphical scene, with all the physics involved (light source, rays, reflections, refractions, color, occlusions etc.).



Implementing it by building packages that include classes for objects involved in creating a graphic scene.

The following packages are included:

- Package Primitives – includes classes for a Coordinate, Point2D, Point3D, Vector, Ray and Material.
- Package Geometries – includes classes for a RadialGeometry, FlatGeometry, Plane,Triangle, Cylinder, Sphere, Rectangle, Quadrangle.
- Package Elements - includes classes for a Camera, Light, LightSource, AmbientLight, DirectonalLight, PointLight, SpotLight.
- Package Scene – Scene.
- Package Renderer.

All the classes above have functions unique to their specific use. More details below.
Everything shown here is implemented through using Java.

בס"ד

Package Primitives

Class Coordinate:

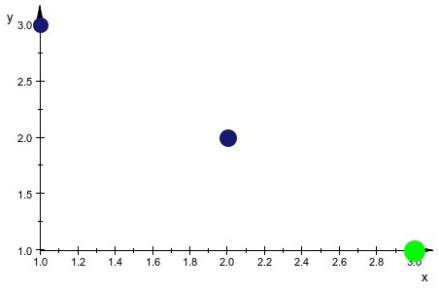
Represents a value on number line.

Functions:

- add (Coordinate other) - returns a new Coordinate the result of Coordinate plus another.
- subtract (Coordinate other) - returns new coordinate result of Coordinate minus the other.
- scale (double num) – returns a new coordinate multiplied by given number.
- multiply (Coordinate other) – returns a new coordinate multiplied by another Coordinate.

Class Point2D:

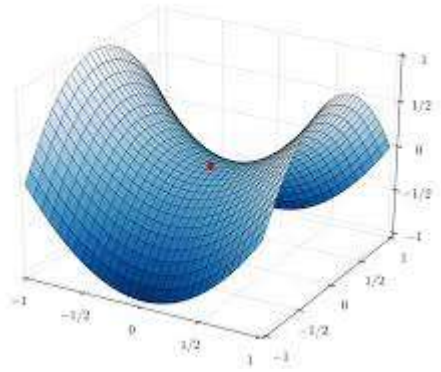
Represents a point in a Two-dimensional space -(x,y)
Contains two Coordinates (x,y).



בס"ד

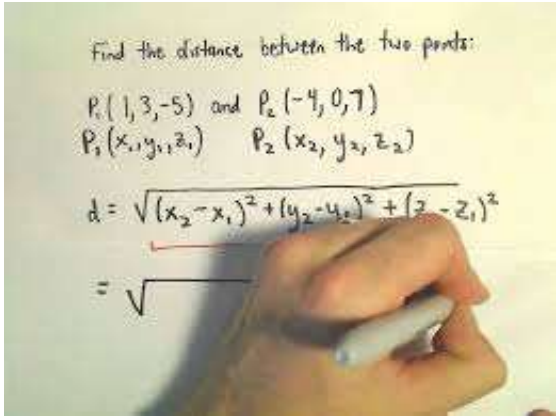
Class Point3D:

Represents a point in a three-dimensional space -(x,y,z).
Contains 3 Coordinates (x,y,z).
Inherits from Point2D and has another Coordinate for z.



Functions:

add (Vector vector) – modifies value of point after adding vector to it.
subtract (Vector vector) - subtracts vector from the point.
distance (Point 3D point) – calculates the distance between two points.



Class Vector:

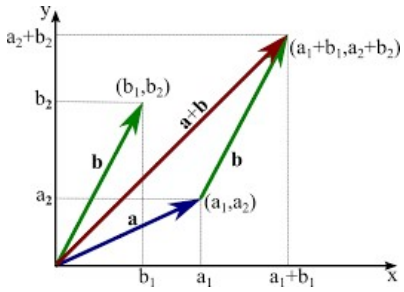
Represents a vector.

בס"ד

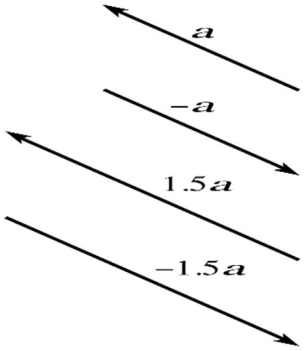
A vector is a quantity that has magnitude (length) and direction. A vector is represented by a directed line segment, a segment with an arrow at one end indicating the direction of movement. Contains a Point3D that is the value for the head of the vector.

Functions:

- add (Vector vector) – modifies the vector by adding another vector.



- subtract (Vector vector) - modifies the vector by subtracting another vector. (the same math as adding but subtract).
- scale (double scalingFactor) - scales the vector by given scaling factor.



- crossProduct(Vector vector) – calculates the cross product calculation between vectors.

בס"ד

$$\vec{u} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \quad \vec{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

$$\vec{u} \times \vec{v} = \begin{bmatrix} u_2v_3 - u_3v_2 \\ u_3v_1 - u_1v_3 \\ u_1v_2 - u_2v_1 \end{bmatrix}$$

- length - calculates the length of the vector.

Length of two-dimensional vector

$$|v| = \sqrt{x^2 + y^2}$$

Length of three-dimensional vector

$$|v| = \sqrt{x^2 + y^2 + z^2}$$

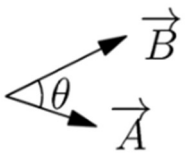
- normalize () – normalizes the vector so length of it will equal 1. This is done by dividing every Coordinate by the length of the vector.
- dotProduct(Vector vector) - calculates dot product between vector and another.

DOT PRODUCT

Definition: $\vec{\mathbf{A}} \cdot \vec{\mathbf{B}} = \sum a_i b_i = a_1 b_1 + a_2 b_2 + a_3 b_3$

This is a scalar.

Geometrically $\vec{\mathbf{A}} \cdot \vec{\mathbf{B}} = |\vec{\mathbf{A}}||\vec{\mathbf{B}}| \cos(\theta)$



Class Ray

בס"ד

A ray is point of origin (3D) and a direction to where it goes. It is implemented by a Point3D for the point of origin, and a vector for the direction it goes.

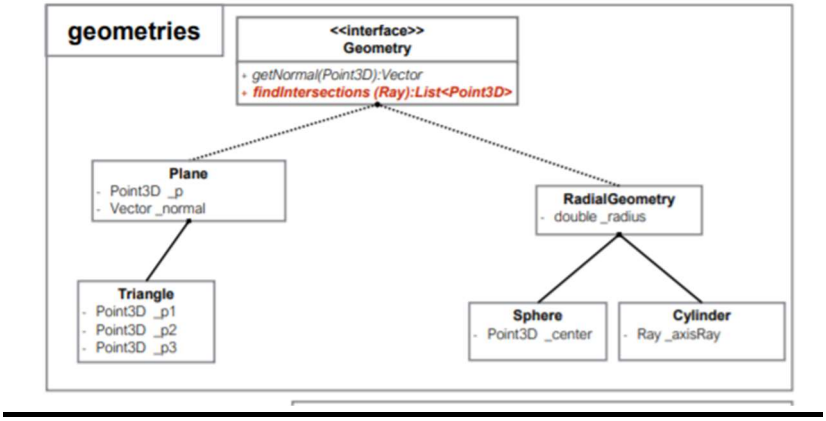
Class Material

represents a type of Material. To do so there are different fields that create the type of the material.
CONTAINS values (all of type Double) for:
_Kd - Diffusion attenuation coefficient
_Ks - Specular attenuation coefficient
_Kr - Reflection coefficient (1 for mirror)
_Kt - Refraction coefficient (1 for transparent)
_n - Refraction index

בס"ד

Package Geometries

Our Architectural Design



package called geometries that contains a collection of classes to describe shapes and Geometric bodies. In order to enable the construction of a scene from a collection of different bodies, the package was built hierarchically. We have defined an abstract class called Geometry for any geometric body and then defined successor classes for the various bodies.

We added an abstract function In the name of getNormal which receives its only parameter on the body and returns the normal (vertical) vector to the body at this point.

We added another abstract class in this package called RadialGeometry, this class inherits from Geometry and includes a field _radius.

בס"ד

Geometric bodies included:

Flat shapes:

- Triangle
- Plane

3D objects:

- Sphere
- Cylinder

Class Geometry

This class is an abstract class called Geometry for any geometric body and then will be defined successor classes for the various bodies.

It includes values for: material, shininess and color of the geometry.

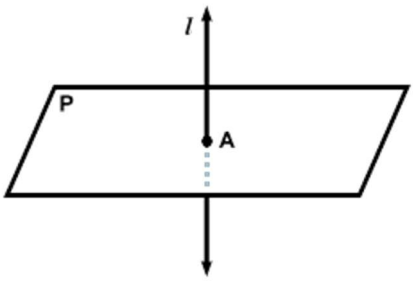
Abstract functions:

- FindIntersections : it will find intersection of a ray and a geometry.
- getNormal: will get the normal of a geometry.

Class Plane

Represents a Plane.

בס"ד



Functions:

- FindIntersections(Ray ray): finds intersection of

a ray and the triangle calculating it as follows:

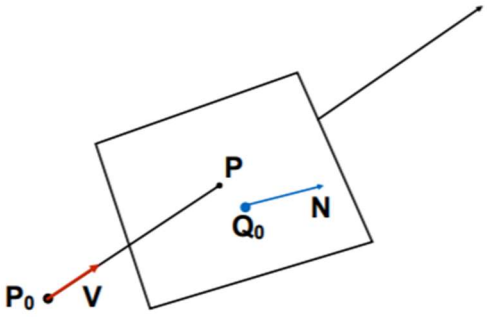
Ray-Plane Intersections

Ray points: $P = P_0 + t \cdot v, \quad t \geq 0$
Plane points: $\mathrm{N} \cdot (Q_0 - P) = 0$

$$N \cdot (Q_0 - t \cdot v - P_0) = 0$$

$$N \cdot (Q_0 - P_0) - t \cdot N \cdot v = 0$$

$$t = \frac{N \cdot (Q_0 - P_0)}{N \cdot v}$$

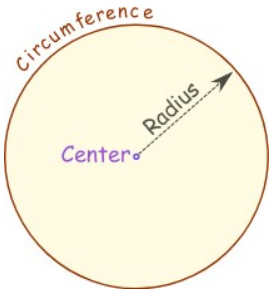


In a manner analogous to the way lines in a two-dimensional space are described using a point-slope form for their equations, planes in a three dimensional space have a natural description using a point in the plane and a vector orthogonal to it (the normal vector) to indicate its "inclination".

Contains a vector normal.
Therefor, the plane contains a point3D and a vector for the normal.

Class RadialGeometry

Abstract class so the geometries that have radius will inherit from.
It has a double value for the radius.



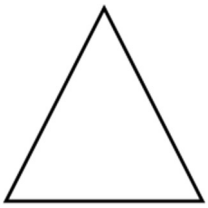
Interface FlatGeometry

This is a interface that unites all the geometries that are flat. Such as triangle and plane.

Class Triangle

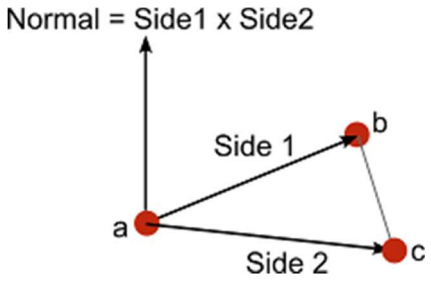
Represents a triangle.

A triangle is a polygon with three edges and three vertices. It is one of the basic shapes in geometry.



Contains 3 Point3D's for the three vertices.
Functions:

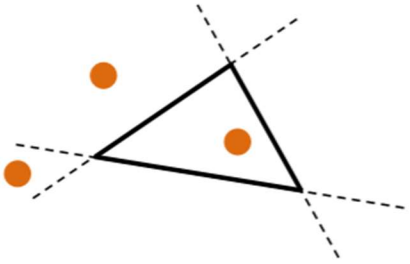
- getNormal: gets the normal of a triangle.
  A surface normal for a triangle can be calculated by taking the vector cross product of two edges of that triangle. The order of the vertices used in the calculation will affect the direction of the normal



- findIntersection: finds point of intersection of ray with the triangle the calculation as follows:

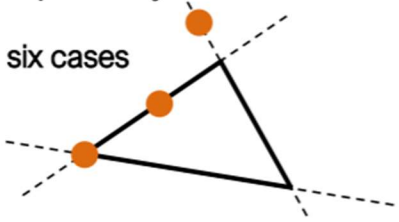
Ray-Triangle intersections
(only if the ray intersects with the plane)

EP: Three cases



VBA:

- One case – ray begins at the plane anywhere
- Three cases (twice) – total six cases
  - The ray begins "before" or **at** the plane



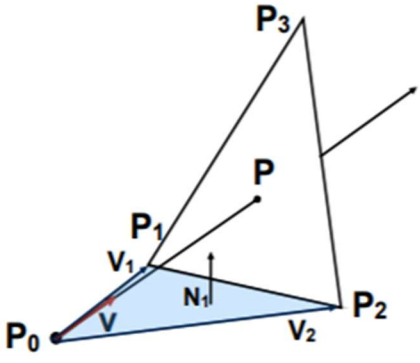
Ray-Triangle Intersections

First, intersect ray with plane (normal is the cross product of the two sides of the triangle).

Then, check if the intersection point is inside triangle

$v_1 = P_1 - P_0$
$v_2 = P_2 - P_0$
$v_3 = P_3 - P_0$
$N_1 = normalize(v_1 \times v_2)$
$N_2 = normalize(v_2 \times v_3)$
$N_3 = normalize(v_3 \times v_1)$

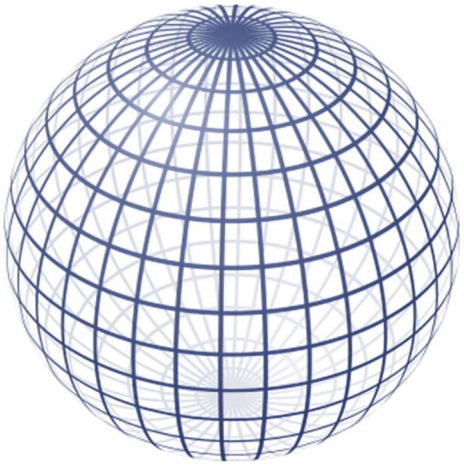
The point is inside if all $(\boldsymbol{P} - \boldsymbol{P_0}) \cdot \boldsymbol{N_i}$ have the same sign (+/-)



בס"ד

Class Sphere

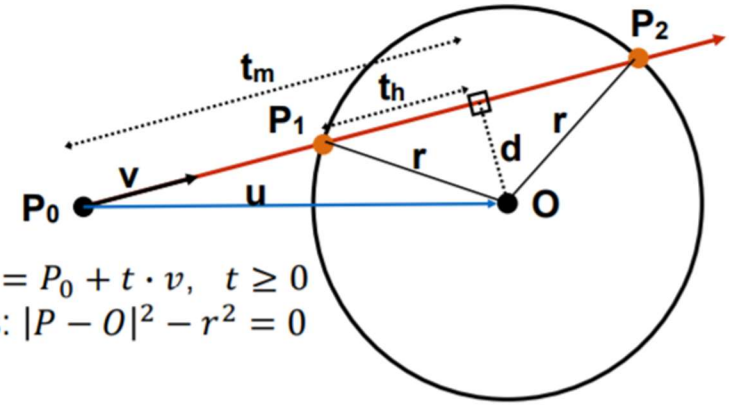
A sphere is a perfectly round geometrical object in three-dimensional space that is the surface of a completely round ball.



It contains a point 3D for the center of it and radius.
It inherits from RadialGeometry.
Functions:

- <u>FindIntersections</u>: finds intersection of ray with the sphere. Doing so by the following calculation

Ray-Sphere Intersections

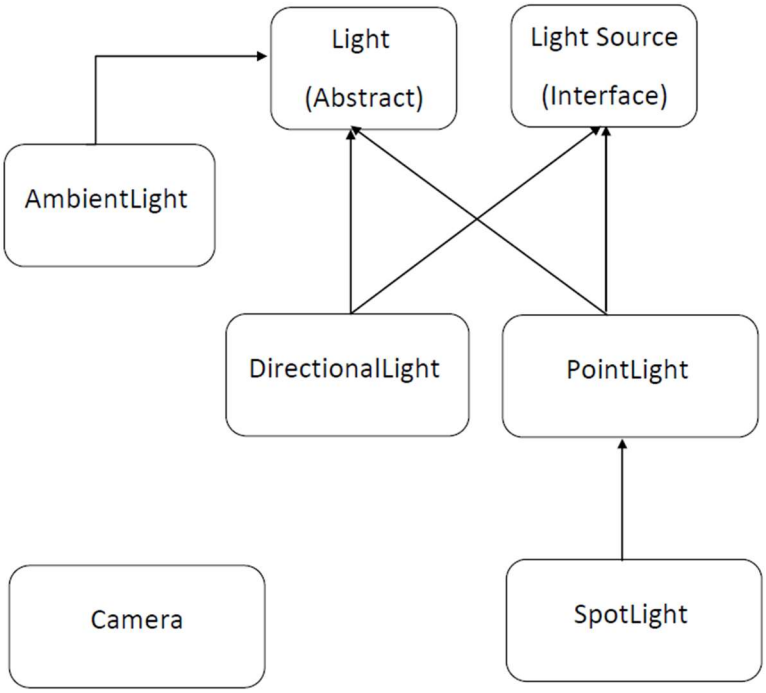


Ray points: $P = P_0 + t \cdot v, \quad t \geq 0$
Sphere points: $|P - O|^2 - r^2 = 0$
$u = O - P_0$
$t_m = v \cdot u$
$d = \sqrt{|u|^2 - {t_m}^2}$ ⇨ if (d>r) there are no intersections
$t_h = \sqrt{r^2 - d^2}$
$t_{1,2} = t_m \pm t_h, \quad P_i = P_0 + t_i \cdot v,$ ⇨ take only $\boldsymbol{t \geq 0}$

- <u>getNormal:</u> returns normal from point on sphere.

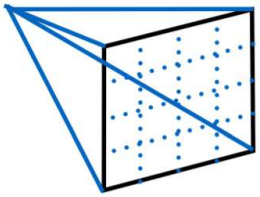
בס"ד

Package Elements



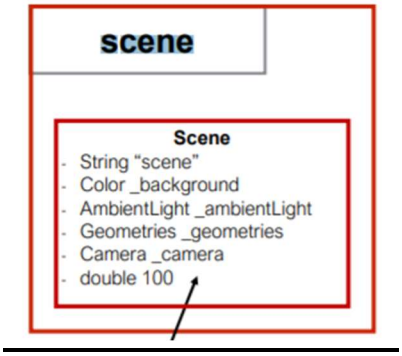
We would like to set up the camera which is our view of our model, of geometric shapes. we start by defining a package of elements. We will define a class for the camera.

Camera



View plane

בס"ד



Class Camera

The camera will contain:

- A three-dimensional point that is the projection center.
- 3 Direction vectors:

  1. VUP - Vector in the positive direction of the system upwards. His continuation in the opposite direction is the vector pointing down.

  2. VRIGHT - A vector in the right-side direction of the system. His continuation in the opposite direction is the vector he points out
  The left side of the system.

  3. VOTOWARD - Vector directed towards the viewing plane

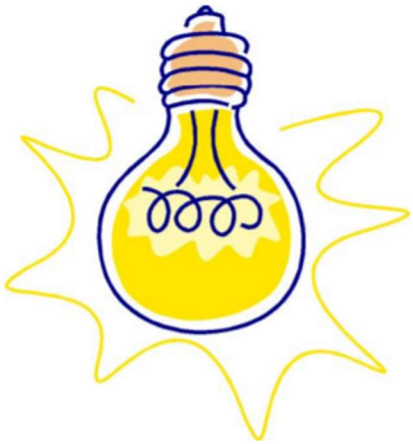
Functions:

- constructRayThroughPixel: returns a ray from the camera

בס"ד

class Light

this class represents the concept of light.
Does the use of the built in color functions of java.



Class AmbientLight

Environmental lighting represents a constant intensity and a constant color light source that affects all objects in the scene evenly.
An ambient light source represents a fixed-intensity and fixed color light source that affects all objects in the scene equally.

$$I_{point3D} = K_{AM} * I_{AM}$$

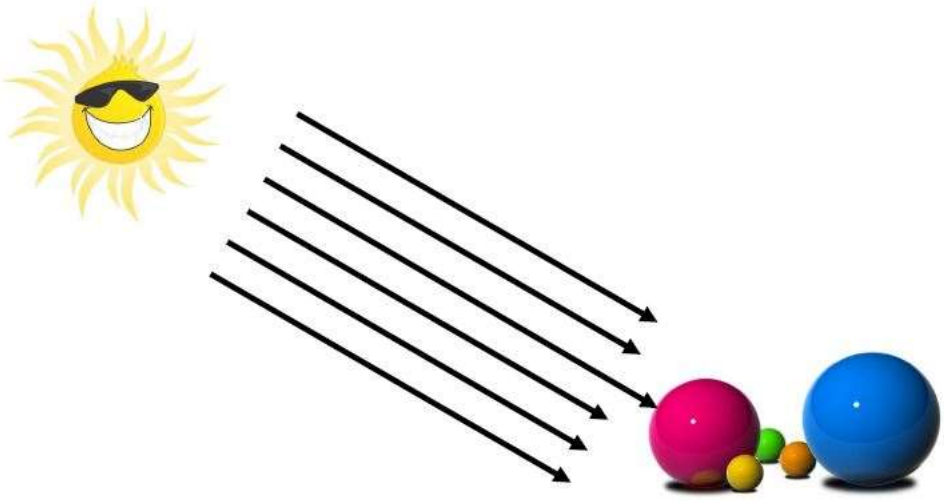
Upon rendering, all objects in the scene are brightened with the specified intensity and color.
Mainly used to provide the scene with a basic view of the different objects in it.

The simplest type of lighting to implement and models how light can be scattered or reflected many times producing a uniform effect.

Class DirectionalLight

Represents a source of light found in infinity, a very, very distant location (like the sun).
A directional light source illuminates all objects equally from a given direction, like an area light of infinite size and infinite distance from the scene;

- intensity (0I)
- Direction (dx, dy, dz) (vector indicating the direction of the projection of light)



- There is no attenuation due to distance (there is no dilution of light due to distance) If so, the power of a directional light source is simply power.

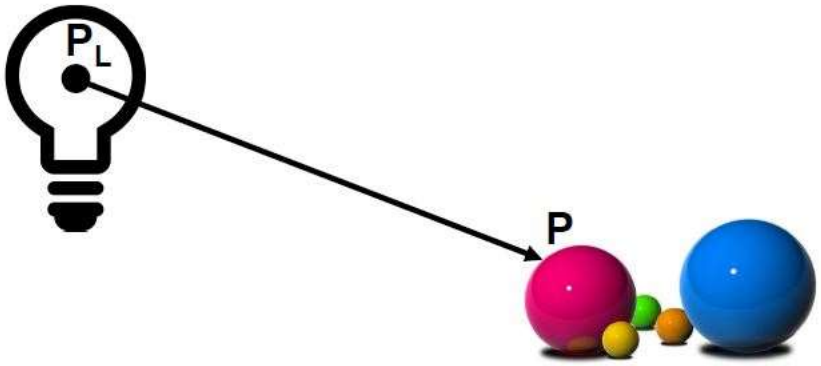
Class PointLight

Modeling a multi-directional light source (such as a light bulb)

- Power (0I)
- A particular location in the scene at point (px, py, pz.)
- This light has no direction
- There are mitigating factors Kc, Kj, and Kq with the distance d (that is, there is thinning of light due to the distance).

If so, the intensity of a dotted light source is equal to the intensity of the partial dilution by distance.

The equation: $I_L = I_0 / (K_c \cdot k_j d \cdot k_q d^2)$

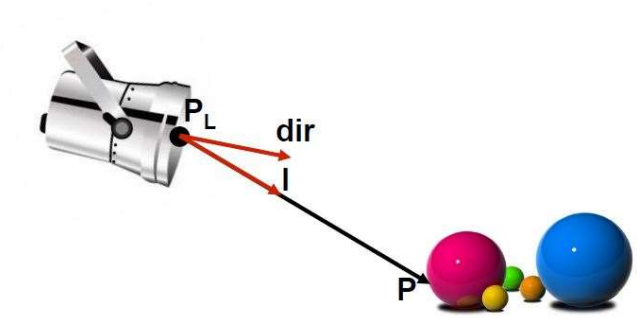


בס"ד

Class spotlight

Represents a Spot Light Source.
Modeling a spot light source with direction (such as the LUXO lamp, the PIXAR animation lamp)

Spot Light Source

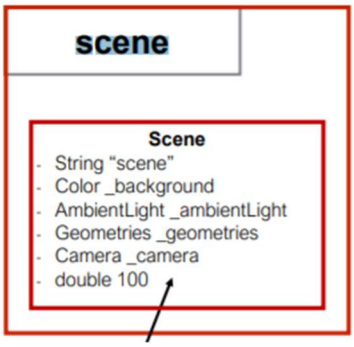


Contains:

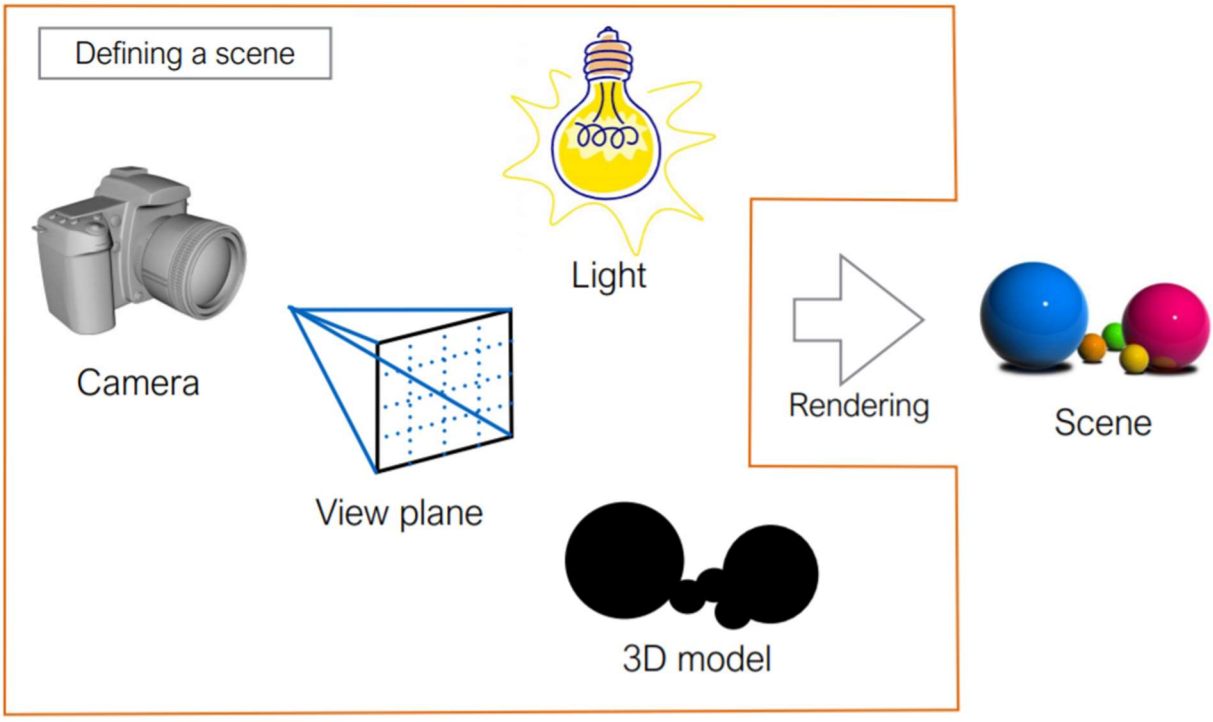
- intensity (Io)
- A particular location in the scene at point (px, py, pz.)
- Direction (dx, dy, dz = D)
- The same attenuation factors as a point light source If so, spot light inherits from point light so its equation will be like a point light, but this time also The effect of the direction of light: the angle between the vector of the light source direction (D) and the vector that hits A larger object (L) will have less light at that point and vice versa. A dot product between the two vectors, both vectors should are normalized.

בס"ד

Package Scene



Until now we have dealt with each body independently, in order to construct a scene consisting of multiple bodies to bind all the components together and save the collection of bodies that describe the scene in the list.



We will add to the project a department called Scene (within a package called Scene) that describes the scene. Fields in this class:

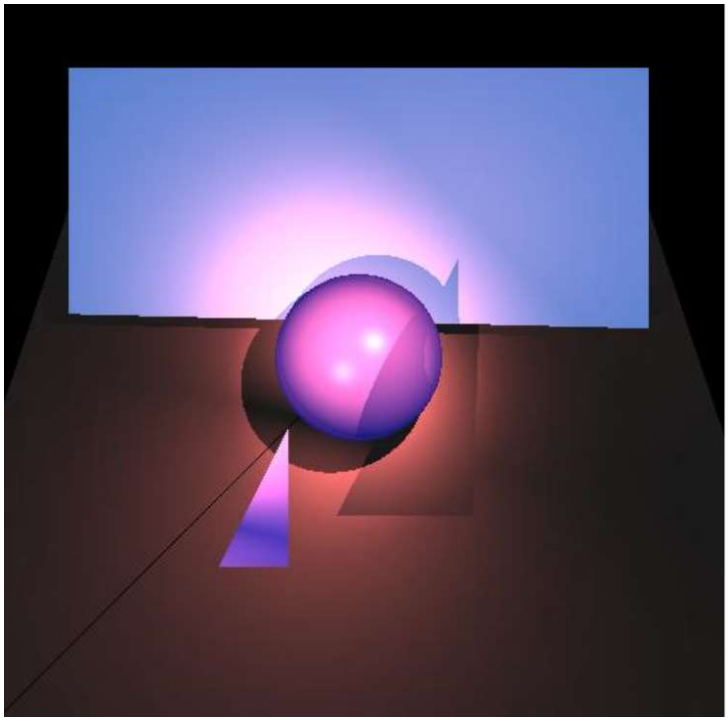
- sceneName - The name of the scene.
- backGround - Background color.
- ambientLight - List of bodies that describe the scene.
- Camera- camera
- screenDistance - The camera distance from the screen.
- geometries - collection of bodies

בס"ד

Package Renderer

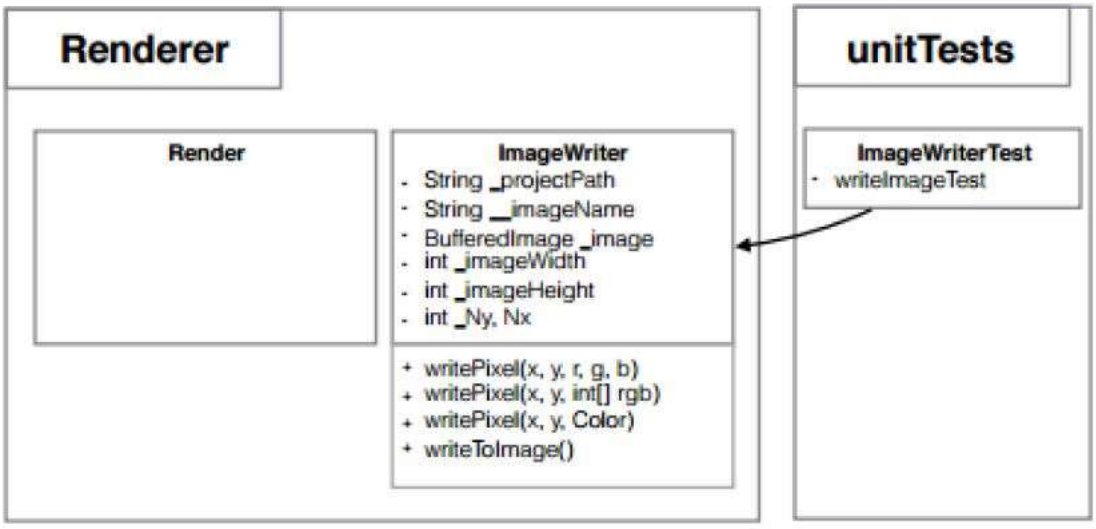
Now we need to implement a class or classes that created the scene itself (with the projection of the horns, finding the color of each pixel, etc.)
And will also be shown on the viewing surface (screen), basically a class that will render.
For this purpose we define a "package" called a renderer that includes two classes:

- Render - does all the actions to create a scene
- ImageWriter - A class that accepts a scene and transfers it to the screen (viewing area).



Class ImageWriter

בס"ד



Implementation of imageWriter:

We'll define fields for the class:

- project name
- Picture name
- The image returned after the process (Buffered image)
- The width of the image
- The length of the image
- The number of squares on the X axis
- The number of squares on the Y axis
  We defined methods of the department:
- The WritePixel function that accepts two pixel indexes (X and Y) and RGB colors

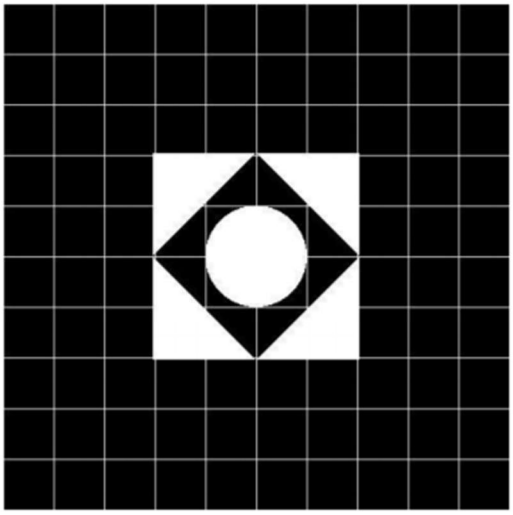
- The WritePixel function accepts two pixel indexes (X and Y) and an RGB array
- The WritePixel function that accepts two pixel indexes (X and Y) and color (color)
- The WriteToImage function returns the ready image

Test Case

Generate the image:

4 triangles
1 sphere

(grid is optional - requires an additional **printGrid(interval)** function)



בס"ד

Class Render

Rendering or image synthesis is the automatic process of generating a photorealistic or non-photorealistic image from a 2D or 3D model by means of computer programs.
This class does all the actions to create the image.
Has these methods and functions:

- **getSceneRayIntersections -**
  The function receives a ray and needs to find its point of intersection with all the geometries in the scene (there is a List of geometries in the Department scene). Returns a list of all the points of intersections.
- **GetClosestPoint -**
  Finds from a list of the closest intersection points the point that the distance from the PO point of projection is minimal.
- **calcColor -**
  Gets a point and calculates the color of the point. Taking into consideration all the possible entities that have impact on the color such as lighting etc. returns the color for given point.
- **printGrid-** prints onto screen a grid (optional).
- **renderImage**- the function implements the image onto the screen.
- **constructRefractedRay-** this function gets a geometry , Point3D and ray and returns a ray that is a refracted after it intersects with the geometry .
- **constructReflectedRay-** this function gets a Vector , Point3D and ray and returns a ray that is a result of reflection after it intersects with the geometry .
- **addColor-** gets two colors and returns the color that is a result of both.
- **calcSpecularComp**- gets these values: double ks, Vector V, Vector N, Vector L,double shininess, Color lightInensity**.**

Returns the color taking into consideration the light returned by the object (depends on the material from which the object is made).

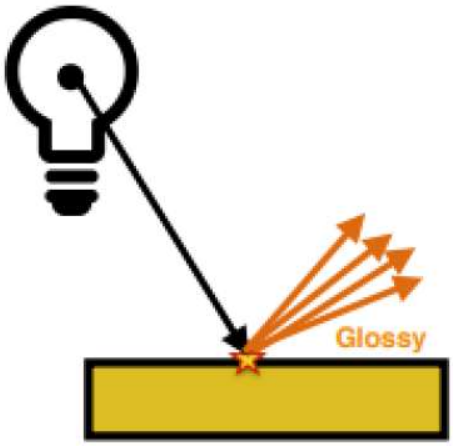
- **CalcDiffusiveComp-** gets these values: double kd, Vector N, Vector L, Color intensityLight.
  Calculates the color thaking into consideration the Diffusion, which means the spread of light on the object.
- **multiColorByNumber**- returns a new color after multiplying it by a scale.
- **Occluded-** gets these values: LightSource light, Point3D point, Geometry geometry. And return a Boolean value to check if the specific point is shadowed.

בס"ד

glossy surfaces

problem:
we came across a issue that glasses completely reflects, which seems not true because there is no dimming of the reflection as the object is further away from the mirror, which happens in real life.

Solution:
Suppose that the surface of the geometry (which is not completely smooth) consists of many tiny aspects (like microfacets.) The norm of each aspect is slightly different in direction and then instead of sending a single reflection ray, it is sent a bit radiate from every aspect (which leads to many rays from one point). Objects that are closer to the mirror will be sharper because the rays are scattered and the rays sent are not yet wide. More distant objects will be more blurry because they are scattered the rays widened. We added this into the code for reflected and refracted rays.



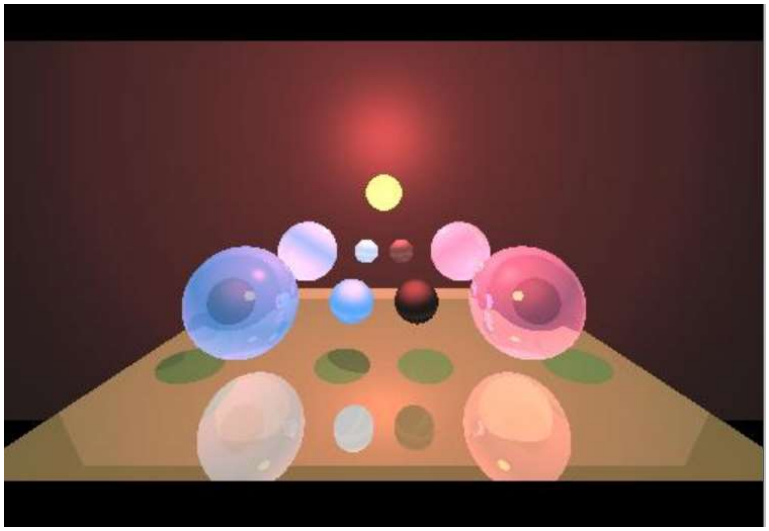
we added within the reflected and refracted ray the addition of taking more rays within a random number (epsilon).

This created rays from camera to the new points (with the epsilon) doing so it solves the problem.

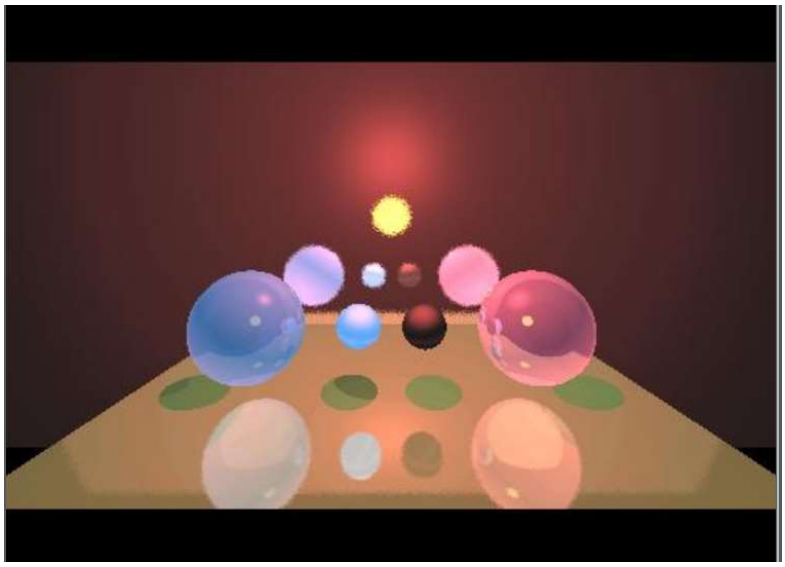
Example:

we created a scene where there is a few geometries. Amongst them were a few sphere. Some are ‘glassy’ where one was In another. In addition we added a mirror and a reflecting floor.

This if before we solved the problem:



This is after:



We can see that the mirror got a little blurry. And so the inner spheres are harder to see.

acceleration

Why accelerate?
Without acceleration, the entire process we did would cause each ray to be sent to have an intersection point with each object in the scene, doing so we allow to actually find the nearest intersection.
so, for a scene with n objects and a picture of I pixels, the complexity will be n * I. If we ignore super-sampling, reflection and translucent rays, then for a scene with a million objects and 2^1024 pixels will be trillion calculations for cropping points between a ray and an object.
And that's very bad.

We solved this issue by putting all the objects in ‘boxes’. We will wrap the objects of the scene in boxes to see if the ray we sent intersects with the box that surrounds the objects. If so, then we will check the

intersection points with the objects in the box. Otherwise we would not bother to check intersection points with the objects.
We did so by creating dynamic boxes that calculate the limit of the objects in them. We preferred to have multiple boxes (instead of only one) so to minimize the calculations.

We found that using this method, the complexity of rendering the scenes were much simpler and quicker and it accelerated the rendering process.

We did so by building another class named Box.

Class box

Contains:

- ArrayList<Geometry> _geometries;//list of geometries
- Limits _limits;//the limits of the box

Methods:

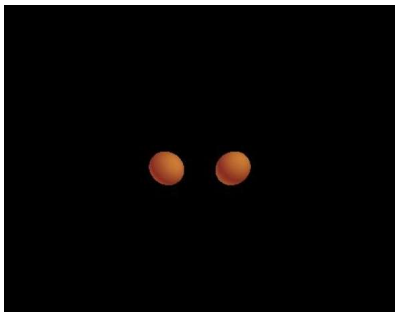
- setLimits() – sets the limits of the box so all the geometries fit in minimal box. Implementing this by getting the maximum and minimum values for x,y,z coordinates of the geometries.
- Intersect – Boolean. It checks if the ray has an intersection with the box.

We added in render a function that uses the boxing method and it receives the list of boxes and it renders the picture by finding first of all intersections with the box and then within the geometries. This saves a lot of unnecessary calculations.

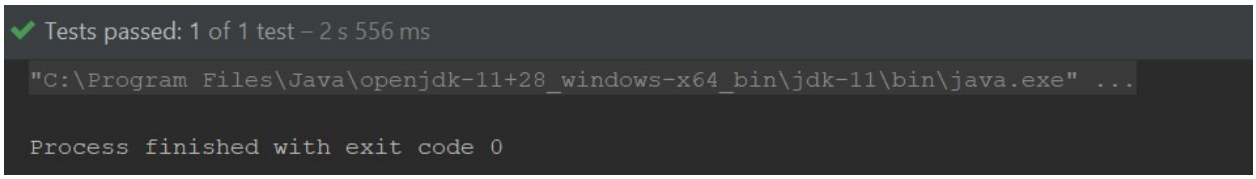
We ran a test that showed the improvement of this:

בס"ד

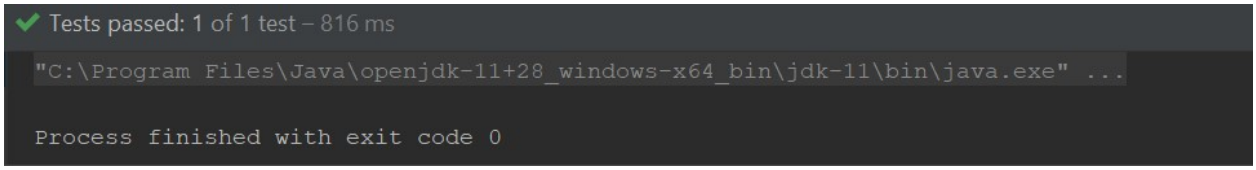
Running time for a specific scene



(very simple scene where there was only 2 spheres) without using the acceleration:



With the acceleration method:



We can see that the running time before acceleration was 2s 556ms And after it was 816 ms.

This is a great improvement for a very simple scene. For a much more complicated scene with numerous geometries and effects the running time will improve tremendously.